Syntax analysis

Syntax analysis is the process of checking the **syntactical structure** of a given input, such as a program or a natural language sentence, according to the **rules of a formal grammar**. It usually involves building a **parse tree** or a **syntax tree** that represents the hierarchical structure of the input

Associativity If an operand has operators on both sides, the side on which the operator takes this operand is decided by the associativity of those operators. If the operation is left-associative, then the operand will be taken by the left operator or if the operation is right-associative, the right operator will take the operand.

Operator Precedence and Associativity

The concept of **operator precedence and associativity in C** helps in determining which operators will be given priority when there are multiple operators in the expression. It is very common to have multiple operators in C language and the compiler first evaluates the operator with higher precedence. It helps to maintain the ambiguity of the expression and helps us in avoiding unnecessary use of parenthesis.
In this article, we will discuss **operator precedence, operator associativity, and precedence table** according to which the priority of the operators in expression is decided in C language.

Operator Precedence and Associativity Table

The following tables list the C operator precedence from highest to lowest and the associativity for each of the operators:

Precedence	Operator	Description	Associativity
1	()	Parentheses (function call)	Left-to-Right
	[]	Array Subscript (Square Brackets)	
	.	Dot Operator	
	->	Structure Pointer Operator	
	++ , —	Postfix increment, decrement	

Precedence	Operator	Description	Associativity
2	**++ / —**	Prefix increment, decrement	Right-to-Left
	+ / –	Unary plus, minus	
	! , ~	Logical NOT, Bitwise complement	
	(type)	Cast Operator	
	*****	Dereference Operator	
	&	Addressof Operator	
	sizeof	Determine size in bytes	
3	***,/,%**	Multiplication, division, modulus	Left-to-Right
4	**+/-**	Addition, subtraction	Left-to-Right
5	**<< , >>**	Bitwise shift left, Bitwise shift right	Left-to-Right
6	**< , <=**	Relational less than, less than or equal to	Left-to-Right
	> , >=	Relational greater than, greater than or equal to	
7	**== , !=**	Relational is equal to, is not equal to	Left-to-Right

Precedence	Operator	Description	Associativity
8	&	Bitwise AND	Left-to-Right
9	^	Bitwise exclusive OR	Left-to-Right
10	|	Bitwise inclusive OR	Left-to-Right
11	&&	Logical AND	Left-to-Right
12	||	Logical OR	Left-to-Right
13	?:	Ternary conditional	Right-to-Left
14	=	Assignment	Right-to-Left
	+= , -=	Addition, subtraction assignment	
	*= , /=	Multiplication, division assignment	
	%= , &=	Modulus, bitwise AND assignment	
	^= , |=	Bitwise exclusive, inclusive OR assignment	
	<<=, >>=	Bitwise shift left, right assignment	
15	,	comma (expression separator)	Left-to-Right

Operator Precedence in C

Operator precedence determines which operation is performed first in an expression with more than one operator with different precedence.

Example of Operator Precedence

Let's try to evaluate the following expression,

10 + 20 * 30

The expression contains two operators, **+ (plus)**, and *** (multiply).** According to the given table, the ***** has higher precedence than **+** so, the first evaluation will be

10 + (20 * 30)

After evaluating the higher precedence operator, the expression is

10 + 600

Now, the + operator will be evaluated.

610

We can verify this using the following C program

- C

```c
// C Program to illustrate operator precedence
#include <stdio.h>

int main()
{
    // printing the value of same expression
    printf("10 + 20 * 30 = %d", 10 + 20 * 30);

    return 0;
}
```

Output

10 + 20 * 30 = 610

As we can see, the expression is evaluated as,**10 + (20 * 30)** but **not as (10 + 20) * 30** due to *** operator** having higher precedence.

Operator Associativity

Operator associativity is used when two operators of the same precedence appear in an expression. Associativity can be either from **Left to Right or Right to Left.**

Example of Operator Associativity

Let's evaluate the following expression,

100 / 5 % 2

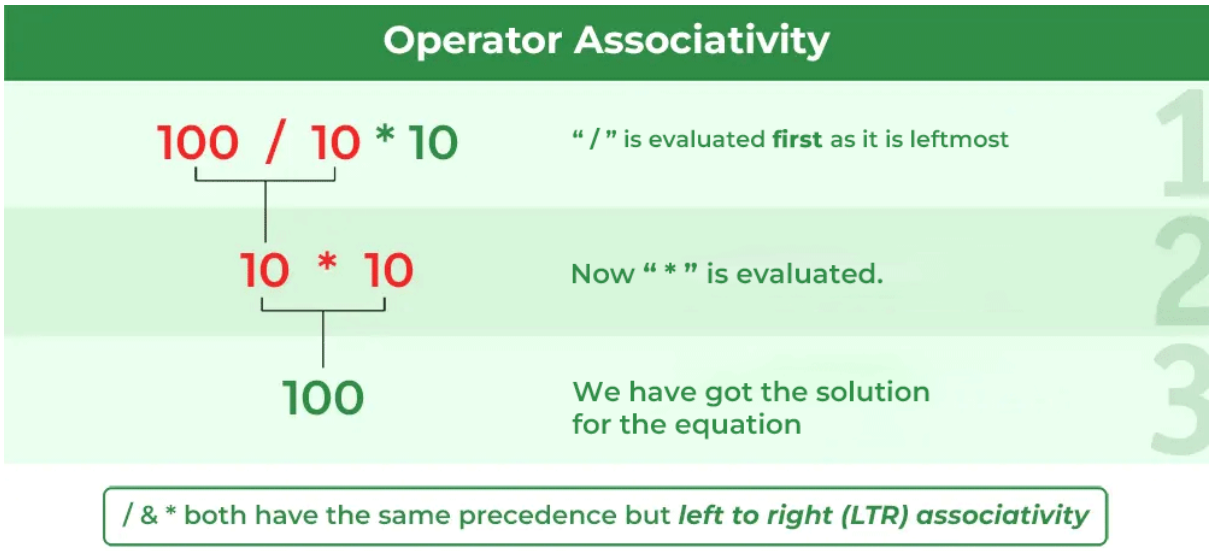
Both / (division) and % (Modulus) operators have the same precedence, so the order of evaluation will be decided by associativity.

According to the given table, the associativity of the multiplicative operators is from **Left to Right.** So,

(100 / 5) % 2
After evaluation, the expression will be

20 % 2
Now, the % operator will be evaluated.

0



We can verify the above using the following C program:

- C

```
// C Program to illustrate operator Associativity
#include <stdio.h>

int main()
{
    // Verifying the result of the same expression
    printf("100 / 5 % 2 = %d", 100 / 5 % 2);

    return 0;
}
```

Output
100 / 5 % 2 = 0

Operators Precedence and Associativity are two characteristics of operators that determine the evaluation order of sub-expressions.

Example of Operator Precedence and Associativity
In general, the concept of precedence and associativity is applied together in expressions. So let's consider an expression where we have operators with various precedence and associativity

exp = 100 + 200 / 10 - 3 * 10

Here, we have four operators, in which the **/** and ***** operators have the same precedence but have higher precedence than the **+** and **–** operators. So, according to the Left-to-Right associativity of **/** and *****, **/** will be evaluated first.

exp = 100 + (200 / 10) - 3 * 10
= 100 + 20 - 3 * 10

After that, ***** will be evaluated,

exp = 100 + 20 - (3 * 10)
= 100 + 20 - 30

Now, between **+** and **–**, **+** will be evaluated due to Left-to-Right associativity.

exp = (100 + 20) - 30
= 120 - 30

At last, **–** will be evaluated.

exp = 120 - 30
= 90

Again, we can verify this using the following C program.

- C

```c
// C Program to illustrate the precedence and associativity
// of the operators in an expression
#include <stdio.h>


int main()
{
    // getting the result of the same expression as the
    // example
    int exp = 100 + 200 / 10 - 3 * 10;
    printf("100 + 200 / 10 - 3 * 10 = %d", exp);


    return 0;
}
```

Output:

100 + 200 / 10 – 3 * 10 = **90**

Important Points

There are a few important points and cases that we need to remember for operator associativity and precedence which are as follows:

1. Associativity is only used when there are two or more operators of the same precedence.

The point to note is associativity doesn't define the order in which operands of a single operator are evaluated. For example, consider the following program,

associativity of the + operator is left to right, but it doesn't mean f1() is always called before f2(). The output of the following program is in-fact compiler-dependent.

Example

- C

```
// Associativity is not used in the below program.
// Output is compiler dependent.

#include <stdio.h>

int x = 0;
int f1()
{
   x = 5;
   return x;
}
int f2()
{
   x = 10;
   return x;
}
int main()
{
   int p = f1() + f2();
   printf("%d ", x);
   return 0;
}
```

Output
10

2. We can use parenthesis to change the order of evaluation

Parenthesis ( ) got the highest priority among all the C operators. So, if we want to change the order of evaluation in an expression, we can enclose that particular operator in **( ) parenthesis** along with its operands.

Example
Consider the given expression

100 + 200 / 10 - 3 * 10
= 90

But if we enclose 100 + 200 in parenthesis, then the result will be different.

(100 + 200) / 10 - 3 * 10 = 0
= 0

As the **+** operator will be evaluated before **/** operator.

2. All operators with the same precedence have the same associativity.

This is necessary, otherwise, there won't be any way for the compiler to decide the evaluation order of expressions that have two operators of the same precedence and different associativity. For example + and – have the same associativity.

3. Precedence and associativity of postfix ++ and prefix ++ are different.
The precedence of postfix ++ is more than prefix ++, their associativity is also different. The associativity of postfix ++ is left to right and the associativity of prefix ++ is right to left. See this for examples.

4. Comma has the least precedence among all operators and should be used carefully.
For example, consider the following program, the output is 1.

Example

- C

```
#include <stdio.h>
int main()
{
    int a;
    a = 1, 2, 3; // Evaluated as (a = 1), 2, 3
    printf("%d", a);
    return 0;
}
```

Output
1

See this and this for more details.

5. There is no chaining of comparison operators in C
In Python, an expression like "c > b > a" is treated as "c > b and b > a", but this type of chaining doesn't happen in C. For example, consider the following program. The output of the following program is "FALSE".

Example

- C

```
#include <stdio.h>
int main()
{
    int a = 10, b = 20, c = 30;

    // (c > b > a) is treated as ((c  > b) > a), associativity of '>'
    // is left to right. Therefore the value becomes ((30 > 20) > 10)
    // which becomes (1 > 10)
    if (c > b > a)
        printf("TRUE");
    else
        printf("FALSE");
```

```
    return 0;
}
```

Output
FALSE

Conclusion
It is necessary to know the precedence and associativity for the efficient usage of operators. It allows us to write clean expressions by avoiding the use of unnecessary parenthesis. Also, it is the same for all the C compilers so it also allows us to understand the expressions in the code written by other programmers.

Also, when confused about or want to change the order of evaluation, we can always rely on parenthesis **( )**. The advantage of brackets is that the reader doesn't have to see the table to find out the order.
Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above.

Transformational grammar

In linguistics, **transformational grammar** (**TG**) or **transformational-generative grammar** (**TGG**) is part of the theory of generative grammar, especially of natural languages. It considers grammar to be a system of rules that generate exactly those combinations of words that form grammatical sentences in a given language and involves the use of defined operations (called **transformations**) to produce new sentences from existing ones. The method is commonly associated with American linguist Noam Chomsky.

Generative algebra was first introduced to general linguistics by the structural linguist Louis Hjelmslev although the method was described before him by Albert Sechehaye in 1908. Chomsky adopted the concept of transformations from his teacher Zellig Harris, who followed the American descriptivist separation of semantics from syntax. Hjelmslev's structuralist conception including semantics and pragmatics is incorporated into functional grammar.

Basic mechanisms

Deep structure and surface structure

The deep structure represents the core semantic relations of a sentence and is mapped onto the surface structure, which follows the phonological form of the sentence very closely, via *transformations*. The concept of transformations had been proposed before the development of deep structure to increase the mathematical and descriptive power of context-free grammars.

Transformations

The usual usage of the term "transformation" in linguistics refers to a rule that takes an input, typically called the deep structure (in the Standard Theory) or D-structure (in the extended standard theory or government and binding theory), and changes it in some restricted way to result in a surface structure (or S-structure). In TG, phrase structure rules generate deep structures. For example, a typical transformation in TG is subject-auxiliary inversion (SAI). That rule takes as its input a declarative sentence with an auxiliary, such as "John has eaten all the heirloom tomatoes", and transforms it into "Has John eaten all the heirloom tomatoes?" In the original formulation (Chomsky 1957), those rules were stated as rules that held over strings of terminals, constituent symbols or both.

X NP AUX Y ⟹ X AUX NP Y

(NP = Noun Phrase and AUX = Auxiliary)

Formal definition

Chomsky's advisor, Zellig Harris, took transformations to be relations between sentences such as "I finally met this talkshow host you always detested" and simpler (kernel) sentences "I finally met this talkshow host" and "You always detested this talkshow host." A transformational-generative (or simply transformational) grammar thus involved two types of productive rules: phrase structure rules, such as "S → NP

VP" (a sentence may consist of a noun phrase followed by a verb phrase) etc., which could be used to generate grammatical sentences with associated parse trees (phrase markers, or P markers); and transformational rules, such as rules for converting statements to questions or active to passive voice, which acted on the phrase markers to produce other grammatically correct sentences. Hjelmslev had called word-order conversion rules "permutations".

In this context, transformational rules are not strictly necessary to generate the set of grammatical sentences in a language, since that can be done using phrase structure rules alone, but the use of transformations provides economy in some cases (the number of rules can be reduced), and it also provides a way of representing the grammatical relations between sentences, which would not be reflected in a system with phrase structure rules alone.[21]

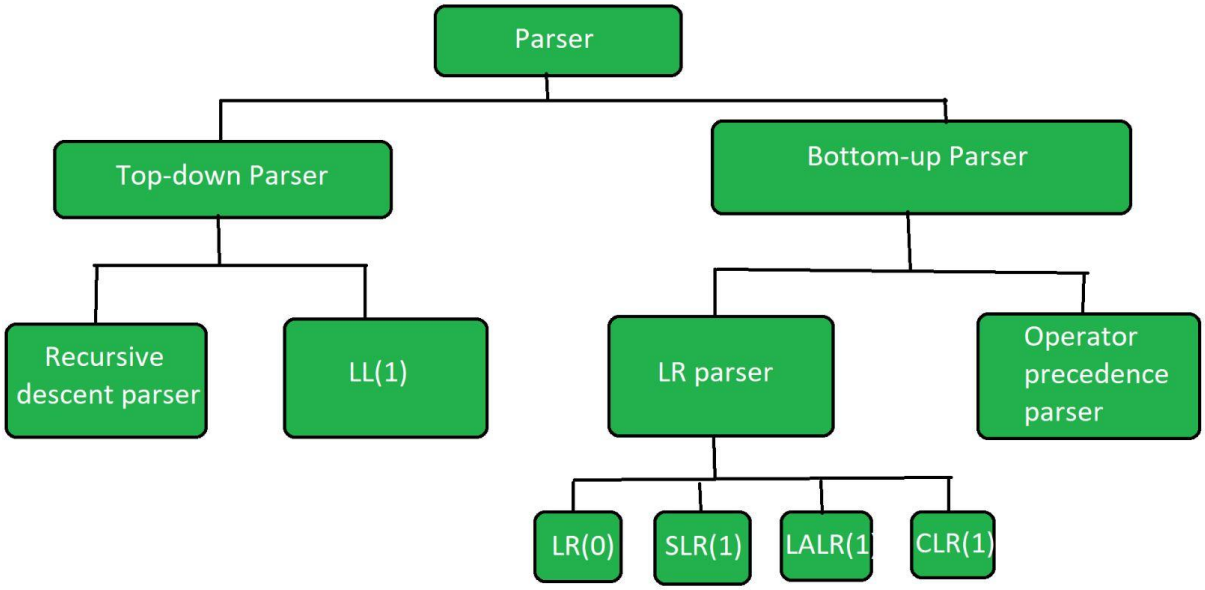
This notion of transformation proved adequate for subsequent versions, including the "extended", "revised extended", and Government-Binding (GB) versions of generative grammar, but it may no longer be sufficient for minimalist grammar, as merge may require a formal definition that goes beyond the tree manipulation characteristic of Move α.

Mathematical representation

An important feature of all transformational grammars is that they are more powerful than context-free grammars. Chomsky formalized this idea in the Chomsky hierarchy. He argued that it is impossible to describe the structure of natural languages with context-free grammars. His general position on the non-context-freeness of natural language has held up since then, though his specific examples of the inadequacy of CFGs in terms of their weak generative capacity were disproved.

Types of Parsers in Compiler Design

The **parser** is that phase of the compiler which takes a token string as input and with the help of existing grammar, converts it into the corresponding Intermediate Representation(IR). The parser is also known as *Syntax Analyzer.*



Classification of Parser

Types of Parser:
The parser is mainly classified into two categories, i.e. Top-down Parser, and Bottom-up Parser. These are explained below:

Top-Down Parser:
The top-down parser is the parser that **generates parse for the given input string** with the help of grammar productions by expanding the non-terminals i.e. it starts from the start symbol and ends on the terminals. It uses left most derivation. Further Top-down parser is classified into 2 types: A recursive descent parser, and Non-recursive descent parser.

1. **Recursive descent parser** is also known as the Brute force parser or the backtracking parser. It basically generates the parse tree by using brute force and backtracking.
2. **Non-recursive descent parser** is also known as LL(1) parser or predictive parser or without backtracking parser or dynamic parser. It uses a parsing table to generate the parse tree instead of backtracking.

Bottom-up Parser:
Bottom-up Parser is the parser that generates the parse tree for the given input string with the help of grammar productions by compressing the terminals i.e. it starts from terminals and ends on the start symbol. It uses the reverse of the rightmost derivation.
Further Bottom-up parser is classified into two types: LR parser, and Operator precedence parser.

Top-down parsing

Top-down parsing in computer science is a parsing strategy where one first looks at the highest level of the parse tree and works down the parse tree by using the rewriting rules of a formal grammar.[1] LL parsers are a type of parser that uses a top-down parsing strategy.

Top-down parsing is a strategy of analyzing unknown data relationships by hypothesizing general parse tree structures and then considering whether the known fundamental structures are compatible with the hypothesis. It occurs in the analysis of both natural languages and computer languages.

Top-down parsing can be viewed as an attempt to find left-most derivations of an input-stream by searching for parse-trees using a top-down expansion of the given formal grammar rules. Inclusive choice is used to accommodate ambiguity by expanding all alternative right-hand-sides of grammar rules.[2]

Simple implementations of top-down parsing do not terminate for left-recursive grammars, and top-down parsing with backtracking may have exponential time complexity with respect to the length of the input for ambiguous CFGs.[3] However, more sophisticated top-down parsers have been created by Frost, Hafiz, and Callaghan,[4][5] which do accommodate ambiguity and left recursion in polynomial time and which generate polynomial-sized representations of the potentially exponential number of parse trees.

Programming language application[edit]

A compiler parses input from a programming language to an internal representation by matching the incoming symbols to production rules. Production rules are commonly defined using Backus–Naur form. An LL parser is a type of parser that does top-down parsing by applying each production rule to the incoming symbols, working from the left-most symbol yielded on a production rule and then proceeding to the next production rule for each non-terminal symbol encountered. In this way the parsing starts on the Left of the result side (right side) of the production rule and evaluates non-terminals from the Left first and, thus, proceeds down the parse tree for each new non-terminal before continuing to the next symbol for a production rule.

For example:

- A ⟶ aBC
- B ⟶ c| cd
- C ⟶ df | eg

which produces the string *A=acdf*

would match A ⟶ aBC and attempt to match B ⟶ c | cd next.
Then C⟶ df | eg would be tried. As one may expect, some languages are more ambiguous than others. For a non-ambiguous language, in which all productions for a non-terminal produce distinct strings, the string produced by one production will not start with the same symbol as the string produced by another production. A non-ambiguous language may be parsed by an LL(1) grammar where the signifies the parser reads ahead one token at a time. For an ambiguous language to be parsed by an LL parser, the parser must lookahead more than 1 symbol, e.g. LL.

The common solution to this problem is to use an LR parser, which is a type of shift-reduce parser, and does bottom-up parsing.

Working of top down parser

In this article, we are going to cover working of top down parser and will see how we can take input and parse it and also cover some basics of top down.

Pre-requisite – Parsing
Top Down Parser :

- In top down technique parse tree constructs from top and input will read from left to right. In top down, In top down parser, It will start symbol from proceed to string.
- It follows left most derivation.
- In top down parser, difficulty with top down parser is if variable contain more than one possibility selecting 1 is difficult.

Working of Top Down Parser :
Let's consider an example where grammar is given and you need to construct a parse tree by using top down parser technique.
Example –

S -> aABe

A -> Abc | b

B -> d

Now, let's consider the input to read and to construct a parse tree with top down approach.

Input –

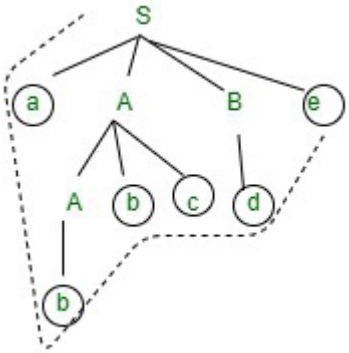
abbcde$

Now, you will see that how top down approach works. Here, you will see how you can generate a input string from the grammar for top down approach.

- First, you can start with S -> a A B e and then you will see input string a in the beginning and e in the end.
- Now, you need to generate ~~a~~bbcd~~e~~ .
- Expand A-> Abc and Expand B-> d.
- Now, You have string like aAbcde and your input string is abbcde.

- Expand A->b.
- Final string, you will get **abbcde**.

Given below is the Diagram explanation for constructing top down parse tree. You can see clearly in the diagram how you can generate the input string using grammar with top down approach.



Recursive Descent Predictive Parsing

In computer science, a **recursive descent parser** is a kind of top-down parser built from a set of mutually recursive procedures (or a non-recursive equivalent) where each such procedure implements one of the nonterminals of the grammar. Thus the structure of the resulting program closely mirrors that of the grammar it recognizes.[1]

A *predictive parser* is a recursive descent parser that does not require backtracking. Predictive parsing is possible only for the class of LL(*k*) grammars, which are the context-free grammars for which there exists some positive integer *k* that allows a recursive descent parser to decide which production to use by examining only the next *k* tokens of input. The LL(*k*) grammars therefore exclude all ambiguous grammars, as well as all grammars that contain left recursion. Any context-free grammar can be transformed into an equivalent grammar that has no left recursion, but removal of left recursion does not always yield an LL(*k*) grammar. A predictive parser runs in linear time.

Recursive descent with backtracking is a technique that determines which production to use by trying each production in turn. Recursive descent with backtracking is not limited to LL(*k*) grammars, but is not guaranteed to terminate unless the grammar is LL(*k*). Even when they terminate, parsers that use recursive descent with backtracking may require exponential time.

Although predictive parsers are widely used, and are frequently chosen if writing a parser by hand, programmers often prefer to use a table-based parser produced by a parser generator, either for an LL(*k*) language or using an alternative parser, such as LALR or LR. This is particularly the case if a grammar is not in LL(*k*) form, as transforming the grammar to LL to make it suitable for predictive parsing is involved. Predictive parsers can also be automatically generated, using tools like ANTLR.

Predictive parsers can be depicted using transition diagrams for each non-terminal symbol where the edges between the initial and the final states are labelled by the symbols (terminals and non-terminals) of the right side of the production rule.

LL(1) Parsing Algorithm

LL(1) Parsing: Here the 1st **L** represents that the scanning of the Input will be done from the Left to Right manner and the second **L** shows that in this parsing technique, we are going to use the Left most Derivation Tree. And finally, the **1** represents the number of look-ahead, which means how many symbols are you going to see when you want to make a decision.

Essential conditions to check first are as follows:

1. The grammar is free from left recursion.
2. The grammar should not be ambiguous.
3. The grammar has to be left factored in so that the grammar is deterministic grammar.

These conditions are necessary but not sufficient for proving a LL(1) parser.

Algorithm to construct LL(1) Parsing Table:

Step 1: First check all the essential conditions mentioned above and go to step 2.

Step 2: Calculate First() and Follow() for all non-terminals.

1. **First():** If there is a variable, and from that variable, if we try to drive all the strings then the beginning Terminal Symbol is called the First.
2. Follow(): What is the Terminal Symbol which follows a variable in the process of derivation.

Step 3: For each production A –> α. (A tends to alpha)

1. Find First(α) and for each terminal in First(α), make entry A –> α in the table.
2. If First(α) contains ε (epsilon) as terminal, then find the Follow(A) and for each terminal in Follow(A), make entry A –> ε in the table.
3. If the First(α) contains ε and Follow(A) contains $ as terminal, then make entry A –> ε in the table for the $.

   To construct the parsing table, we have two functions:

In the table, rows will contain the Non-Terminals and the column will contain the Terminal Symbols. All the **Null Productions** of the Grammars will go under the Follow elements and the remaining productions will lie under the elements of the First set.

Now, let's understand with an example.

Example 1: Consider the Grammar:

```
E --> TE'
E' --> +TE' | ε
T --> FT'
T' --> *FT' | ε
F --> id | (E)

*ε denotes epsilon
```

Step 1: The grammar satisfies all properties in step 1.

Step 2: Calculate first() and follow().

Find their First and Follow sets:

	First	Follow
E –> TE’	{ id, (}	{ $,) }
E’ –> +TE’/ ε	{ +, ε }	{ $,) }
T –> FT’	{ id, (}	{ +, $,) }
T’ –> *FT’/ ε	{ *, ε }	{ +, $,) }
F –> id/(E)	{ id, (}	{ *, +, $,) }

Step 3: Make a parser table.
Now, the LL(1) Parsing Table is:

	id	+	*	(	)	$
E	E –> TE’			E –> TE’		
E’		E’ –> +TE’			E’ –> ε	E’ –> ε
T	T –> FT’			T –> FT’		
T’		T’ –> ε	T’ –> *FT’		T’ –> ε	T’ –> ε
F	F –> id			F –> (E)		

As you can see that all the null productions are put under the Follow set of that symbol and all the remaining productions lie under the First of that symbol.
Note: Every grammar is not feasible for LL(1) Parsing table. It may be possible that one cell may contain more than one production.

Bottom up Parsing

Bottom-up parsing is a technique to analyze the syntax of a text by starting from the smallest units (terminals) and combining them into larger structures (non-terminals) until the start symbol of the grammar is reached. Some examples of bottom-up parsers are LR, LALR and CYK parsers.

Bottom-up **parser:**

- It will start from string and proceed to start.
- In Bottom-up parser, Identifying the correct handle (substring) is always difficult.
- It will follow rightmost derivation in reverse order.

Note:
In bottom-up parser, no variable that's why not have any derivation from the bottom but in reverse order it is looking like top-down, when you have rightmost derivation.

Working of Bottom-up parser :
Let's consider an example where grammar is given and you need to construct a parse tree by using bottom-up parser technique.
Example –

```
S -> aABe
A -> Abc | b
B -> d
```

Now, let's consider the input to read and to construct a parse tree with bottom-up approach.

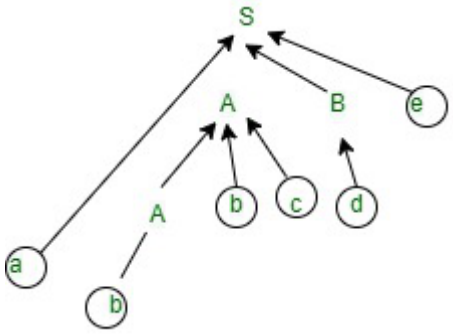
Input –

```
abbcde$
```

Now, you will see that how bottom-up approach works. Here, you will see how you can generate a input string from the grammar for bottom-up approach.

- First, you can start with A -> b.
- Now, expand A -> Abc.
- After that Expand B-> d.
- In the last, just expand the S -> aABe
- Final string, you will get **abbcde**.

Given below is the Diagram explanation for constructing bottom-up parse tree. You can see clearly in the diagram how you can generate the input string using grammar with bottom-up approach.



From the above explanation and diagram, you can clearly see and can say it follows reverse of the rightmost derivation.

LR Parser

LR parsing is one type of bottom up parsing. It is used to parse the large class of grammars.

In the LR parsing, "L" stands for left-to-right scanning of the input.

"R" stands for constructing a right most derivation in reverse.

"K" is the number of input symbols of the look ahead used to make number of parsing decision.

LR parsing is divided into four parts: LR (0) parsing, SLR parsing, CLR parsing and LALR parsing.

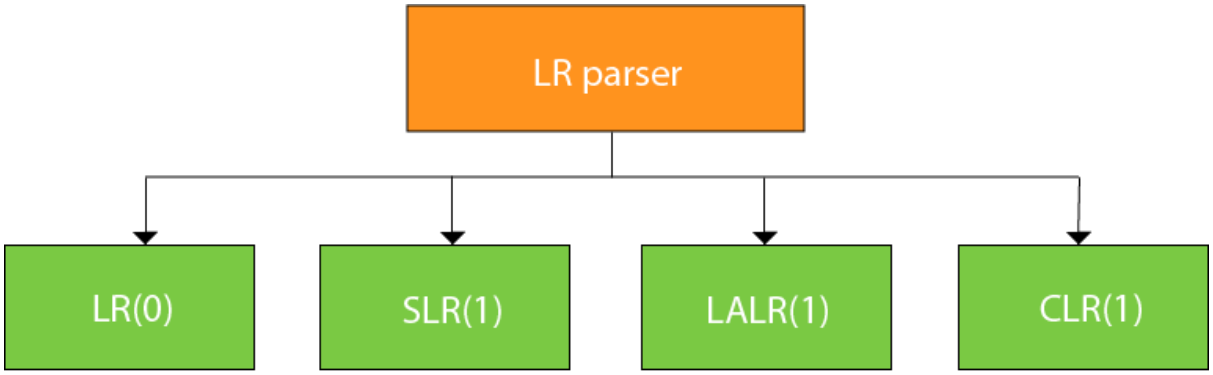


Fig: Types of LR parser

LR algorithm:

The LR algorithm requires stack, input, output and parsing table. In all type of LR parsing, input, output and stack are same but parsing table is different.

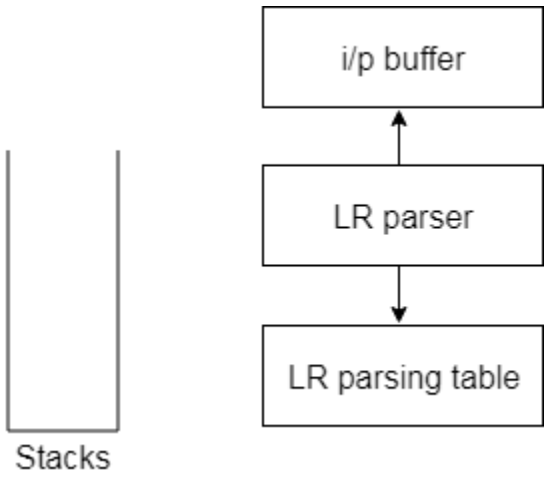


Fig: Block diagram of LR parser

Input buffer is used to indicate end of input and it contains the string to be parsed followed by a $ Symbol.

A stack is used to contain a sequence of grammar symbols with a $ at the bottom of the stack.

Parsing table is a two-dimensional array. It contains two parts: Action part and Go To part.

LR (1) Parsing

Various steps involved in the LR (1) Parsing:

- For the given input string write a context free grammar.
- Check the ambiguity of the grammar.
- Add Augment production in the given grammar.
- Create Canonical collection of LR (0) items.
- Draw a data flow diagram (DFA).
- Construct a LR (1) parsing table.

Augment Grammar

Augmented grammar G` will be generated if we add one more production in the given grammar G. It helps the parser to identify when to stop the parsing and announce the acceptance of the input.

Example

Given grammar

1. S → AA
2. A → aA | b

The Augment grammar G` is represented by

1. S`→ S
2. S → AA
3. A → aA | b

LALR(1) Parser

In computer science, an **LALR parser**[a] or **Look-Ahead, Left-to-right, Rightmost Derivation parser** is part of the compiling process where human readable text is converted into computer instructions. An LALR parser is a software tool to process (parse) code into a very specific internal representation that the compiler can work from. This happens according to a set of production rules specified by a formal grammar for a computer language.

An LALR parser is a simplified version of a canonical LR parser.

The LALR parser was invented by Frank DeRemer in his 1969 PhD dissertation, *Practical Translators for LR(k) languages*, in his treatment of the practical difficulties at that time of implementing LR(1) parsers. He showed that the LALR parser has more language recognition power than the LR(0) parser, while requiring the same number of states as the LR(0) parser for a language that can be recognized by both parsers. This makes the LALR parser a memory-efficient alternative to the LR(1) parser for languages that are LALR. It was also proven that there exist LR(1) languages that are not LALR. Despite this weakness, the power of the LALR parser is sufficient for many mainstream computer languages, including Java, though the reference grammars for many languages fail to be LALR due to being ambiguous.

The original dissertation gave no algorithm for constructing such a parser given a formal grammar. The first algorithms for LALR parser generation were published in 1973.[4] In 1982, DeRemer and Tom Pennello published an algorithm that generated highly memory-efficient LALR parsers.[5] LALR parsers can be automatically generated from a grammar by an LALR parser generator such as Yacc or GNU Bison. The automatically generated code may be augmented by hand-written code to augment the power of the resulting parser.

The LALR(1) parser is less powerful than the LR(1) parser, and more powerful than the SLR(1) parser, though they all use the same production rules. The simplification that the LALR parser introduces consists in merging rules that have identical **kernel item sets**, because during the LR(0) state-construction process the lookaheads are not known. This reduces the power of the parser because not knowing the lookahead symbols can confuse the parser as to which grammar rule to pick next, resulting in **reduce/reduce conflicts**. All conflicts that arise in applying a LALR(1) parser to an unambiguous LR(1) grammar are reduce/reduce conflicts. The SLR(1) parser performs further merging, which introduces additional conflicts.

The standard example of an LR(1) grammar that cannot be parsed with the LALR(1) parser, exhibiting such a reduce/reduce conflict, is:

```
S → a E c
  → a F d
  → b F c
  → b E d
E → e
F → e
```

In the LALR table construction, two states will be merged into one state and later the lookaheads will be found to be ambiguous. The one state with lookaheads is:

```
E → e. {c,d}
F → e. {c,d}
```

An LR(1) parser will create two different states (with non-conflicting lookaheads), neither of which is ambiguous. In an LALR parser this one state has conflicting actions (given lookahead c or d, reduce to E or F), a "reduce/reduce conflict"; the above grammar will be declared ambiguous by a LALR parser generator and conflicts will be reported.

To recover, this ambiguity is resolved by choosing E, because it occurs before F in the grammar. However, the resultant parser will not be able to recognize the valid input sequence `b e c`, since the ambiguous sequence `e c` is reduced to `(E → e) c`, rather than the correct `(F → e) c`, but `b E c` is not in the grammar.